Floating-Point IEEE Filter for Microsoft* Windows* 2000 on the Intel® Itanium™ Architecture

Application Note

March 2001

Contents

1. Introduction 4

2. Data Types 6

2.1. Overview 6

2.2. Data Structures 6

3. Examples 11

3.1. Example 1. Floating-Point Exceptions Raised by Scalar Instructions from Assembly Language Code 11

3.2. Example 2. Floating-Point Exceptions Raised by Scalar Instructions from C Language Code 23

3.3. Example 3. Floating-Point Exceptions Raised by Parallel Instructions from Assembly Language Code 25

4. Authors 31

5. Acknowledgments 31

6. References 31

Revision History

Rev.	Draft/Changes	Date
—001	• Initial Release	March 2001

1. Introduction

The Floating-Point IEEE Filter for the Intel® Itanium™ processor is a function, `_fpieee_flt()`, that is provided to help in handling unmasked floating-point exceptions for applications running on the Itanium architecture. Its name refers to the IEEE Standard for Binary Floating-Point Arithmetic, IEEE-Std 754-1985 [1], which defines some of the entities used by the IEEE filter function: rounding modes, precision, floating-point formats, floating-point operations, and floating-point exceptions.

The IEEE filter decodes the excepting instruction and re-arranges the floating-point exception information provided by the operating system in a format easier to interpret by a user program. For example, in the case of parallel floating-point instructions, the user is presented only with the component(s) that raised unmasked floating-point exception(s). The other component (if any) is handled in the background by the IEEE Filter. For every unmasked floating-point exception, the IEEE filter invokes a user defined floating-point exception handler, which passes back to the IEEE filter the desired result for the excepting instruction before execution is continued.

In the Itanium architecture, four status fields are provided in the Floating-Point Status Register (FPSR, or Application Register 40), that contain control and status bits. The Floating-Point IEEE Filter handles unmasked floating-point exceptions raised by instructions that employ the user status field, or sf0 (status field 0).

For information on Itanium processor floating-point exceptions see *The Itanium™ Architecture Software Developer's Manual* [2], *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic* [3], or for more details, *Itanium™ Processor Floating-point Software Assistance and Floating-point Exception Handling* [4] (Chapter 1 - Introduction, Chapter 2 – Software Assistance Faults and Traps on the Itanium™ Processor, and Chapter 3 – Conditions Causing, and Responses to Floating-Point Exceptions).

Purpose of `_fpieee_flt`

Invokes a user-defined trap handler for IEEE floating-point exceptions (also for denormal exceptions in this implementation)

Prototype

```
int _fpieee_flt (
    unsigned long exc_code,
    struct _EXCEPTION_POINTERS *exc_info,
    int handler(_FPIEEE_RECORD *) );
```

Required header: `fpieee.h`

Return Value

The return value of `_fpieee_flt` is the value returned by the user handler, unless the exception is not recognized or the associated instruction cannot be decoded, in which case `EXCEPTION_CONTINUE_SEARCH` is returned (see below). Thus, the IEEE filter routine may be used in the except clause of a structured exception-handling (SEH) `_try/_except` construct.

The return code of the IEEE floating-point filter can be:

EXCEPTION_CONTINUE_EXECUTION - the floating-point exception was successfully handled, and a result is being provided; this value is returned only if it is also the return value from the user floating-point exception handler to the IEEE filter

EXCEPTION_CONTINUE_SEARCH - the floating-point exception handling was unsuccessful; this assumes that another exception handler should be searched for

EXCEPTION_EXECUTE_HANDLER – the code following the _except clause in the _try/_except construct is executed; next, code following the _try/_except (if any) is executed

Parameters

exc_code - exception code

exc_info - -pointer to the Windows* 2000 exception information structure

handler - pointer to user's IEEE trap-handler routine

Remarks

The _fpieee_flt function invokes a user-defined trap handler for IEEE floating-point exceptions and provides it with all relevant information. This routine serves as an exception filter in the SEH mechanism, which invokes the user’s own IEEE exception handler when necessary.

Note

Because _fpieee_flt() accesses the _FP128 data type defined in fpieee.h using ldf.fill and stf.spill instructions, all variables of this type must be 16-byte aligned . Therefore, make sure that this type is defined as

```
typedef struct __declspec(align(16)) {
      unsigned long W[4];
} _FP128;
```

which ensures its proper alignment (otherwise unaligned access faults might be generated).

2. Data Types

2.1. Overview

The `_FPIEEE_RECORD` structure, defined in `fpieee.h`, contains information pertaining to an IEEE floating-point exception. This structure is passed to the user-defined trap handler by _fpieee_flt.

_FPIEEE_RECORD Field	**Description**
`Unsigned int RoundingMode,` `unsigned int Precision`	These fields contain information on the floating-point environment at the time the exception occurred.
`Unsigned int Operation`	Indicates the type of operation that caused the trap. For example, if the type is a comparison (`_FpCodeCompare`), the user can supply one of the special `_FPIEEE_COMPARE_RESULT` values (as defined in `fpieee.h`) in the `Result.Value` field. The conversion type (`_FpCodeConvert`) indicates that the trap occurred during a floating-point conversion operation. The user can look at the `Operand1` and `Result` types to determine the type of conversion
`Cause, Enable, Status`	Fields indicating the cause of the exception, which floating-point exceptions are enabled (unmasked), and the values of the floating-point status bits

`_FPIEEE_VALUE Operand1,`
`_FPIEEE_VALUE Operand2,`
`_FPIEEE_VALUE Operand3,`
`_FPIEEE_VALUE Result`
	These structures indicate the types and values of the result and operands, through the following fields:
	`OperandValid` - flag indicating whether the corresponding value is valid.
	`Format` - data type of the corresponding value, which may be returned even if the corresponding value is not valid;
	`Value` - result or operand data value.

2.2. Data Structures

The IEEE filter receives the exception information from the operating system kernel through the first two parameters: unsigned long `exc_code`, and `struct _EXCEPTION_POINTERS *exc_info`. The user does not need to know about this if the IEEE filter is employed to handle unmasked floating-point exceptions, but will need all the details otherwise (i.e. if floating-point exceptions are handled directly by the user handler, without calling the IEEE filter function).

The exception code `exc_code` can be one of the following:

```
STATUS_FLOAT_INVALID_OPERATION,
STATUS_FLOAT_DIVIDE_BY_ZERO,
STATUS_FLOAT_DENORMAL_OPERAND,
STATUS_FLOAT_INEXACT_RESULT,
STATUS_FLOAT_OVERFLOW,
STATUS_FLOAT_UNDERFLOW,
STATUS_FLOAT_MULTIPLE_FAULTS,
STATUS_FLOAT_MULTIPLE_TRAPS,
```

all defined in `winnt.h` (the last two types correspond to multiple faults or traps raised by parallel Itanium processor instructions). This value is provided by the `GetExceptionCode()` function.

The data structure passed to the IEEE filter by means of the second parameter has its type defined also in `winnt.h`:

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT ContextRecord;
} _EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

where the exception record and the context record are defined also in `winnt.h`. A pointer to this data structure is provided by the `GetExceptionInformation()` function.

The following information is needed by the user who invokes the IEEE filter to handle unmasked floating-point exceptions. It refers to data types defined in `fpieee.h`, file which has to be included by every application that links in the IEEE floating-point filter function..

The IEEE filter and the user handler exchange information through the IEEE record:

```
typedef struct {
  unsigned int RoundingMode : 2;
  unsigned int Precision : 3;
  unsigned int Operation :12;
  FPIEEE_EXCEPTION_FLAGS Cause;
  FPIEEE_EXCEPTION_FLAGS Enable;
  FPIEEE_EXCEPTION_FLAGS Status;
  FPIEEE_VALUE Operand1;
  FPIEEE_VALUE Operand2;
  FPIEEE_VALUE Result;
  FPIEEE_VALUE Operand3;
} _FPIEEE_RECORD, *_PFPIEEE_RECORD;
```

All the fields except one (`Result`) are completely filled in by the IEEE filter function, which thus communicates the necessary exception information to the user handler. The `Result` field is used to pass information both ways – from the IEEE filter to the user handler, and back. The IEEE filter function ensures that this value, when provided by the user handler, will be used as the result of the exception instruction when execution of the user application is continued. The types and formats of the various fields in the IEEE record are specified below. Programming examples are also included at the end.

The type of the `RoundingMode` field is defined by the following 'enum'. The four entries correspond to the four IEEE rounding modes for floating-point operations: rounding to nearest, to negative infinity, to positive infinity, and toward zero:

```
typedef enum {
  FpRoundNearest,
  FpRoundMinusInfinity,
  FpRoundPlusInfinity,
```

```
  FpRoundChoppe;
} _FPIEEE_ROUNDING_MODE;
```

The `Precision` field has the following type, which contains entries for the most common IEEE floating-point data types: single precision (with 24-bit significands), double precision (with 53-bit significands), double-extended precision (with 64-bit significands), and quad precision (with 113-bit significands). In addition, one entry is provided for the 'full' precision type on the particular system:

```
typedef enum {
    _FpPrecisionFull,
    _FpPrecision53,
    _FpPrecision24,
    _FpPrecision64,
    _FpPrecision113
} _FPIEEE_PRECISION;
```

The `Operation` field defines the possible floating-point operations that can raise unmasked exceptions. Besides add, subtract, multiply, divide, square root, compare, conversion from floating-point to integer, conversion with truncation, and some mathematical functions introduced with the IEEE filter for IA-32, this list includes also entries for three-operand instructions from the floating-point multiply-add family defined in the Itanium architecture. These operations are `fma` (calculates a * b + c with only one rounding error), `fms` (calculates a * b – c), and `fnma` (calculates –a * b + c). Separate entries are present for `fma.s`, `fms.s`, `fnma.s` (single precision specified statically), `fma.d`, `fms.d`, `fnma.d` (double precision specified statically), and `fma`, `fms`, `fnma` (the precision is specified dynamically by fields of the Floating-Point Status Register). The other Itanium processor instructions that have entries defined in the enumeration below are `fmax` (floating-point maximum), `fmin` (floating-point minimum), `famax` (floating-point absolute maximum), and `famin` (floating-point absolute minimum).

```
typedef enum {
    _FpCodeUnspecified,
    _FpCodeAdd,
    _FpCodeSubtract,
    _FpCodeMultiply,
    _FpCodeDivide,
    _FpCodeSquareRoot,
    _FpCodeRemainder,
     FpCodeCompare,
    _FpCodeConvert,
    _FpCodeRound,
    _FpCodeTruncate,
    _FpCodeFloor,
    _FpCodeCeil,
    _FpCodeAcos,
     … // transcendental or other functions
    _FpCodeNegate,
    _FpCodeFmin,
    _FpCodeFmax,
    _FpCodeConvertTrunc,
    _XMMIAddps,
    _XMMIAddss,
     … // other  SSE/SSE2 instructions
    _FpCodeFma,
    _FpCodeFmaSingle,
    _FpCodeFmaDouble,
    _FpCodeFms,
    _FpCodeFmsSingle,
    _FpCodeFmsDouble,
    _FpCodeFnma,
    _FpCodeFnmaSingle,
```

```
    _FpCodeFnmaDouble,
    _FpCodeFamin,
    _FpCodeFamax
} _FP_OPERATION_CODE;
```

The next three fields in the IEEE record have all the type `_FPIEEE_EXCEPTION_FLAGS`, defined below. These fields are `Cause`, `Enable`, and `Status`. The `Cause` field specifies the cause of the floating-point exception. This will correspond to an IEEE-defined floating-point fault (invalid operation or divide-by-zero), or to a floating-point trap (overflow, underflow, or inexact result). Note that the denormal exceptions (floating-point faults) that can be raised by Itanium processor (as well as IA-32) floating-point instructions, are not defined by the IEEE standard. Therefore, it is implicitly assumed that a denormal exception was raised whenever the user-defined exception handler is invoked by the IEEE filter with no `Cause` bit set. The `Enable` field specifies which of the five IEEE floating-point exceptions are enabled (or unmasked). The `Status` field specifies whether the five IEEE floating-point status flags are set or not. Note that the IEEE record does not specify whether the denormal exceptions are enabled or not, nor the value of the denormal status flag.

The user may change the `Enable` bits, so that a different subset of floating-point exceptions will be masked and/or unmasked after returning from the user handler to the IEEE filter, and then back to the user application. Note though that on the Itanium processor, there is no need for the user to clear the exception status flags after handling a floating-point exception (unlike in IA-32 for FPU instructions, where leaving a status flag set while the corresponding exception is unmasked, will trigger another floating-point exception on the execution of the following non-waiting floating-point instruction).

```
typedef struct {
    unsigned int Inexact : 1;
    unsigned int Underflow : 1;
    unsigned int Overflow : 1;
    unsigned int ZeroDivide : 1;
    unsigned int InvalidOperation : 1;
} _FPIEEE_EXCEPTION_FLAGS;
```

The last four fields in the IEEE record correspond to the excepting floating-point instruction operands (up to three – `Operand1`, `Operand2`, and `Operand3`), and to its result (the `Result` field). The operand information is passed by the IEEE filter to the user floating-point exception handler. For the result, the `OperandValid` and `Format` fields are always passed by the IEEE filter to the user handler. The information in the `value` field depends on the type of floating-point exception. For floating-point faults, no information is passed in this field from the IEEE filter to the user handler, which may fill in the desired value before returning to the IEEE filter. For floating-point traps, a result value is provided to the user handler by the IEEE filter (as specified by the IEEE Standard for Binary Floating-Point Arithmetic), but the user handler may also set a new value in this field before returning to the IEEE filter. Note that for unmasked underflow or overflow, the result is scaled up or down (respectively) by the IEEE filter: by 2^{192} for single precision, by 2^{1536} for double precision, by 2^{24576} for double-extended precision, and by 2^{98304} for floating-point register file format (with 17-bit exponent). All four entries for operands and result have the type `_FPIEEE_VALUE` defined as shown below.

```
typedef struct {
    union {
        _FP32        Fp32Value;
        _FP64        Fp64Value;
        _FP80        Fp80Value;
        _FP128       Fp128Value;
        _I16         I16Value;
        _I32         I32Value;
        _I64         I64Value;
        _U16         U16Value;
```

```
        _U32         U32Value;
        _U64         U64Value;
        _BCD80       Bcd80Value;
        char         *StringValue;
        int          CompareValue;
    } Value;
    unsigned int OperandValid : 1;
    unsigned int Format : 4;
} _FPIEEE_VALUE;
```

Some of the value field types from `_FPIEEE_VALUE` are also defined in `fpieee.h`. These are `_FP32/_FP64/_FP80/_FP128` for single/double/double-extended/register file precision floating-point values, `_I16/_I32/_I64` for 16/32/64-bit integers, `_U16/_U32/_U64` for unsigned 16/32/64-bit integers, and `_BCD80` for the BCD values:

```
typedef float _FP32;
      typedef double _FP64;
      typedef struct {
        unsigned short W[5];
}_FP80;
typedef struct __declspec(align(16)) {
        unsigned long W[4];
 } _FP128;
 typedef short _I16;
      typedef int  _I32;
      typedef struct {
        unsigned long W[2];
}_I64;
typedef unsigned short  _U16;
typedef unsigned int  _U32;
typedef struct {
        unsigned long W[2];
} _U64;
typedef struct {
        unsigned short W[5];
 } _BCD80;
```

When the `Value` field of the `_FPIEEE_VALUE` `data type` is of type `CompareValue`, it can be used to specify one of four possible values for the result of a compare operation: equal, less than, greater than, or unordered, as specified in the following enum:

```
typedef enum {
  FpCompareEqual,
  FpCompareGreater,
  FpCompareLess,
  FpCompareUnordered
} _FPIEEE_COMPARE_RESULT;
```

Finally, the `Format` field has type `_FPIEEE_FORMAT`:

```
typedef enum {
_FpFormatFp32,
_FpFormatFp64,
_FpFormatFp80,
_FpFormatFp128,
_FpFormatI16,
_FpFormatI32,
_FpFormatI64,
```

```
_FpFormatU16,
_FpFormatU32,
_FpFormatU64,
_FpFormatBcd80,
_FpFormatCompare,
_FpFormatString,
_FpFormatFp82
} _FPIEEE_FORMAT;
```

3. Examples

The following three examples illustrate raising and handling unmasked floating-point exceptions using the Microsoft* Structured Exception Handling mechanism (SEH) in Windows 2000, running on the Itanium processor.

The first example illustrates floating-point exceptions raised by scalar (non-parallel) floating-point instrructions from assembly language routines, called from C code. The second example is similar, but contains only C code. The third example illustrates exceptions raised by parallel floating-point instrructions from assembly language routines, called from C code.

Note that the preferred method of reading or writing floating-point status and control information is by using the _statusfp(), _clearfp(), and _controlfp() functions. The following examples achieve the same result by reading and writing directly the FPSR.

3.1. Example 1. Floating-Point Exceptions Raised by Scalar Instructions from Assembly Language Code

The next example illustrates how to use the IEEE filter to handle unmasked floating-point exceptions raised by Itanium processor instructions. The user handler is `fpieee_handler()`, which covers the most common cases of floating-point exceptions that can occur in Itanium processor applications. Function `main()` is presented next. It initializes the operand values and the FPSR value for the last function presented below, `run_fma()`, which is used to trigger the five possible exceptions for the `fma.s` instruction: invalid operation, denormal operand, overflow, underflow, and inexact operation. Triggering floating-point exceptions could be done directly from C, as seen further in Example 2.

```
int fpieee_handler (_FPIEEE_RECORD *ieee) {

float f32;
double f64;
_FP80 f80;
_FP128 f82;
int i32;
```

```
__int64 i64;
unsigned int u32;
unsigned __int64 u64;

printf ("\n *** BEGIN USER HANDLER ***\n\n");

if (ieee->RoundingMode == _FpRoundNearest)
  printf ("USER HANDLER: round to nearest\n");
else if (ieee->RoundingMode == _FpRoundMinusInfinity)
  printf ("USER HANDLER: round to negative infinity\n");
else if (ieee->RoundingMode == _FpRoundPlusInfinity)
  printf ("USER HANDLER: round to positive infinity\n");
else if (ieee->RoundingMode == _FpRoundChopped)
  printf ("USER HANDLER: round to zero\n");
else
  printf ("USER HANDLER: unknown rounding mode\n");


if (ieee->Precision == _FpPrecision24)
  printf ("USER HANDLER: single precision\n");
else if (ieee->Precision == _FpPrecision53)
  printf ("USER HANDLER: double precision\n");
else if (ieee->Precision == _FpPrecision64)
  printf ("USER HANDLER: double-extended precision\n");
else
  printf ("USER HANDLER: unknown precision mode\n");

if (ieee->Operation == _FpCodeAdd)
  printf ("USER HANDLER: add operation\n");
else if (ieee->Operation == _FpCodeSubtract)
  printf ("USER HANDLER: subtract operation\n");
 else if (ieee->Operation == _FpCodeMultiply)
  printf ("USER HANDLER: multiply operation\n");
else if (ieee->Operation == _FpCodeDivide)
  printf ("USER HANDLER: divide operation\n");
else if (ieee->Operation == _FpCodeSquareRoot)
  printf ("USER HANDLER: square root operation\n");
else if (ieee->Operation == _FpCodeCompare)
  printf ("USER HANDLER: compare operation\n");
else if (ieee->Operation == _FpCodeConvert)
  printf ("USER HANDLER: convert operation\n");
else if (ieee->Operation == _FpCodeConvertTrunc)
  printf ("USER HANDLER: convert and truncate operation\n");
else if (ieee->Operation == _FpCodeFma)
  printf ("USER HANDLER: floating-point multiply-add  operation\n");
else if (ieee->Operation == _FpCodeFmaSingle)
  printf ("USER HANDLER: single floating-point multiply-add operation\n");
else if (ieee->Operation == _FpCodeFmaDouble)
  printf ("USER HANDLER: double floating-point multiply-add  operation\n");
else if (ieee->Operation == _FpCodeFms)
```

```
  printf ("USER HANDLER: floating-point multiply-subtract  operation\n");
else if (ieee->Operation == _FpCodeFmsSingle)
  printf ("USER HANDLER: single floating-point multiply-subtract "
     "operation\n");
else if (ieee->Operation == _FpCodeFmsDouble)
  printf ("USER HANDLER: double floating-point multiply-subtract "
     "operation\n");
else if (ieee->Operation == _FpCodeFnma)
  printf ("USER HANDLER: negative floating-point multiply-add "
     "operation\n");
else if (ieee->Operation == _FpCodeFnmaSingle)
  printf ("USER HANDLER: single negative floating-point multiply-add "
     "operation\n");
else if (ieee->Operation == _FpCodeFnmaDouble)
  printf ("USER HANDLER: negative double floating-point multiply-add "
     "operation\n");
else if (ieee->Operation == _FpCodeFmin)
  printf ("USER HANDLER: floating-point minimum operation\n");
else if (ieee->Operation == _FpCodeFmax)
  printf ("USER HANDLER: floating-point maximum operation\n");
else if (ieee->Operation == _FpCodeFamin)
  printf ("USER HANDLER: floating-point absolute minimum operation\n");
else if (ieee->Operation == _FpCodeFamax)
  printf ("USER HANDLER: floating-point absolute maximum operation\n");
else if (ieee->Operation == _FpCodeUnspecified)
  printf ("USER HANDLER: unspecified floating-point operation\n");
else
  printf ("USER HANDLER: unknown floating-point operation\n");

printf ("USER HANDLER: cause bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->Cause.Inexact, ieee->Cause.Underflow, ieee->Cause.Overflow,
    ieee->Cause.ZeroDivide, ieee->Cause.InvalidOperation);

printf ("USER HANDLER: enable bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->Enable.Inexact, ieee->Enable.Underflow,
    ieee->Enable.Overflow, ieee->Enable.ZeroDivide,
    ieee->Enable.InvalidOperation);

printf ("USER HANDLER: status bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->Status.Inexact, ieee->Status.Underflow,
    ieee->Status.Overflow, ieee->Status.ZeroDivide,
    ieee->Status.InvalidOperation);

if (ieee->Operand1.OperandValid) {
  printf ("USER HANDLER: operand1 is valid\n");
  if (ieee->Operand1.Format == _FpFormatFp32) {
    printf ("USER HANDLER: operand1 has format _FpFormatFp32\n");
    f32 = ieee->Operand1.Value.Fp32Value;
    printf ("USER HANDLER: operand1 value = %8.8x\n", *(unsigned int *)&f32);
  } else if (ieee->Operand1.Format == _FpFormatFp64) {
```

```
      printf ("USER HANDLER: operand1 has format _FpFormatFp64\n");
      f64 = ieee->Operand1.Value.Fp64Value;
      printf ("USER HANDLER: operand1 value = %I64x\n", *(unsigned __int64 *)&f64);
    } else if (ieee->Operand1.Format == _FpFormatFp80) {
      printf ("USER HANDLER: operand1 has format _FpFormatFp80\n");
      f80 = ieee->Operand1.Value.Fp80Value;
      printf ("USER HANDLER: operand1 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
          f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
    } else if (ieee->Operand1.Format == _FpFormatFp82) {
      printf ("USER HANDLER: operand1 has format _FpFormatFp82\n");
      f82 = ieee->Operand1.Value.Fp128Value;
      printf ("USER HANDLER: operand1 value = %8.8x%8.8x%8.8x%8.8x\n",
          f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
    } else if (ieee->Operand1.Format == _FpFormatI32) {
      printf ("USER HANDLER: operand1 has format _FpFormatI32\n");
      i32 = ieee->Operand1.Value.I32Value;
      printf ("USER HANDLER: operand1 value = %8.8x\n", i32);
    } else if (ieee->Operand1.Format == _FpFormatI64) {
      printf ("USER HANDLER: operand1 has format _FpFormatI64\n");
      i64 = (((__int64)ieee->Operand1.Value.I64Value.W[1]) << 32) |
          (unsigned __int64)ieee->Operand1.Value.I64Value.W[0];
      printf ("USER HANDLER: operand1 value = %I64x\n", i64);
    } else if (ieee->Operand1.Format == _FpFormatU32) {
      printf ("USER HANDLER: operand1 has format _FpFormatU32\n");
      u32 = ieee->Operand1.Value.U32Value;
      printf ("USER HANDLER: operand1 value = %8.8x\n", u32);
    } else if (ieee->Operand1.Format == _FpFormatU64) {
      printf ("USER HANDLER: operand1 has format _FpFormatU64\n");
      u64 = (((unsigned __int64)ieee->Operand1.Value.U64Value.W[1]) << 32) |
          (unsigned __int64)ieee->Operand1.Value.U64Value.W[0];
      printf ("USER HANDLER: operand1 value = %I64x\n", u64);
    } else if (ieee->Operand1.Format == _FpFormatCompare) {
      printf ("USER HANDLER: operand1 has format _FpFormatCompare "
          "(invalid)\n");
    } else {
      printf ("USER HANDLER: operand1 has unknown format\n");
    }
  } else {
    printf ("USER HANDLER: operand1 is not valid\n");
  }

  if (ieee->Operand2.OperandValid) {
    printf ("USER HANDLER: operand2 is valid\n");
    if (ieee->Operand2.Format == _FpFormatFp32) {
      printf ("USER HANDLER: operand2 has format _FpFormatFp32\n");
      f32 = ieee->Operand2.Value.Fp32Value;
      printf ("USER HANDLER: operand2 value = %8.8x\n", *(unsigned int *)&f32);
    } else if (ieee->Operand2.Format == _FpFormatFp64) {
      printf ("USER HANDLER: operand2 has format _FpFormatFp64\n");
      f64 = ieee->Operand2.Value.Fp64Value;
```

```
    printf ("USER HANDLER: operand2 value = %I64x\n", (unsigned __int64 *)&f64);
  } else if (ieee->Operand2.Format == _FpFormatFp80) {
    printf ("USER HANDLER: operand2 has format _FpFormatFp80\n");
    f80 = ieee->Operand2.Value.Fp80Value;
    printf ("USER HANDLER: operand2 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
  } else if (ieee->Operand2.Format == _FpFormatFp82) {
    printf ("USER HANDLER: operand2 has format _FpFormatFp82\n");
    f82 = ieee->Operand2.Value.Fp128Value;
    printf ("USER HANDLER: operand2 value = %8.8x%8.8x%8.8x%8.8x\n",
        f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
  } else if (ieee->Operand2.Format == _FpFormatI32) {
    printf ("USER HANDLER: operand2 has format _FpFormatI32\n");
    i32 = ieee->Operand2.Value.I32Value;
    printf ("USER HANDLER: operand2 value = %8.8x\n", i32);
  } else if (ieee->Operand2.Format == _FpFormatI64) {
    printf ("USER HANDLER: operand2 has format _FpFormatI64\n");
    i64 = (((__int64)ieee->Operand2.Value.I64Value.W[1]) << 32) |
        (unsigned __int64)ieee->Operand2.Value.I64Value.W[0];
    printf ("USER HANDLER: operand2 value = %I64x\n", i64);
  } else if (ieee->Operand2.Format == _FpFormatU32) {
    printf ("USER HANDLER: operand2 has format _FpFormatU32\n");
    u32 = ieee->Operand2.Value.U32Value;
    printf ("USER HANDLER: operand2 value = %8.8x\n", u32);
  } else if (ieee->Operand2.Format == _FpFormatU64) {
    printf ("USER HANDLER: operand2 has format _FpFormatU64\n");
    u64 = (((unsigned __int64)ieee->Operand2.Value.U64Value.W[1]) << 32) |
        (unsigned __int64)ieee->Operand2.Value.U64Value.W[0];
    printf ("USER HANDLER: operand2 value = %I64x\n", u64);
  } else if (ieee->Operand2.Format == _FpFormatCompare) {
    printf ("USER HANDLER: operand2 has format _FpFormatCompare "
        "(invalid)\n");
  } else {
    printf ("USER HANDLER: operand2 has unknown format\n");
  }
} else {
  printf ("USER HANDLER: operand2 is not valid\n");
}

if (ieee->Operand3.OperandValid) {
  printf ("USER HANDLER: operand3 is valid\n");
  if (ieee->Operand3.Format == _FpFormatFp32) {
    printf ("USER HANDLER: operand3 has format _FpFormatFp32\n");
    f32 = ieee->Operand3.Value.Fp32Value;
    printf ("USER HANDLER: operand3 value = %8.8x\n", *(unsigned int *)&f32);
  } else if (ieee->Operand3.Format == _FpFormatFp64) {
    printf ("USER HANDLER: operand3 has format _FpFormatFp64\n");
    f64 = ieee->Operand3.Value.Fp64Value;
    printf ("USER HANDLER: operand3 value = %I64x\n", (unsigned __int64 *)&f64);
  } else if (ieee->Operand3.Format == _FpFormatFp80) {
```

```
      printf ("USER HANDLER: operand3 has format _FpFormatFp80\n");
      f80 = ieee->Operand3.Value.Fp80Value;
      printf ("USER HANDLER: operand3 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
          f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
    } else if (ieee->Operand3.Format == _FpFormatFp82) {
      printf ("USER HANDLER: operand3 has format _FpFormatFp82\n");
      f82 = ieee->Operand3.Value.Fp128Value;
      printf ("USER HANDLER: operand3 value = %8.8x%8.8x%8.8x%8.8x\n",
          f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
    } else if (ieee->Operand3.Format == _FpFormatI32) {
      printf ("USER HANDLER: operand3 has format _FpFormatI32\n");
      i32 = ieee->Operand3.Value.I32Value;
      printf ("USER HANDLER: operand3 value = %8.8x\n", i32);
    } else if (ieee->Operand3.Format == _FpFormatI64) {
      printf ("USER HANDLER: operand3 has format _FpFormatI64\n");
      i64 = (((__int64)ieee->Operand3.Value.I64Value.W[1]) << 32) |
          (unsigned __int64)ieee->Operand3.Value.I64Value.W[0];
      printf ("USER HANDLER: operand3 value = %I64x\n", i64);
    } else if (ieee->Operand3.Format == _FpFormatU32) {
      printf ("USER HANDLER: operand3 has format _FpFormatU32\n");
      u32 = ieee->Operand3.Value.U32Value;
      printf ("USER HANDLER: operand3 value = %8.8x\n", u32);
    } else if (ieee->Operand3.Format == _FpFormatU64) {
      printf ("USER HANDLER: operand3 has format _FpFormatU64\n");
      u64 = (((unsigned __int64)ieee->Operand3.Value.U64Value.W[1]) << 32) |
          (unsigned __int64)ieee->Operand3.Value.U64Value.W[0];
      printf ("USER HANDLER: operand3 value = %I64x\n", u64);
    } else if (ieee->Operand3.Format == _FpFormatCompare) {
      printf ("USER HANDLER: operand3 has format _FpFormatCompare "
          "(invalid)\n");
    } else {
      printf ("USER HANDLER: operand3 has unknown format\n");
    }
  } else {
    printf ("USER HANDLER: operand3 is not valid\n");
  }

  if (ieee->Result.OperandValid) {
    printf ("USER HANDLER: result is valid\n");
    if (ieee->Result.Format == _FpFormatFp32) {
      printf ("USER HANDLER: result has format _FpFormatFp32\n");
      f32 = ieee->Result.Value.Fp32Value;
      printf ("USER HANDLER: result value = %8.8x\n", *(unsigned int *)&f32);
    } else if (ieee->Result.Format == _FpFormatFp64) {
      printf ("USER HANDLER: result has format _FpFormatFp64\n");
      f64 = ieee->Result.Value.Fp64Value;
      printf ("USER HANDLER: result value = %I64x\n", (unsigned __int64 *)&f64);
    } else if (ieee->Result.Format == _FpFormatFp80) {
      printf ("USER HANDLER: result has format _FpFormatFp80\n");
      f80 = ieee->Result.Value.Fp80Value;
```

```
    printf ("USER HANDLER: result value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
  } else if (ieee->Result.Format == _FpFormatFp82) {
    printf ("USER HANDLER: result has format _FpFormatFp82\n");
    f82 = ieee->Result.Value.Fp128Value;
    printf ("USER HANDLER: result value = %8.8x%8.8x%8.8x%8.8x\n",
        f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
  } else if (ieee->Result.Format == _FpFormatI32) {
    printf ("USER HANDLER: result has format _FpFormatI32\n");
    i32 = ieee->Result.Value.I32Value;
    printf ("USER HANDLER: result value = %8.8x\n", i32);
  } else if (ieee->Result.Format == _FpFormatI64) {
    printf ("USER HANDLER: result has format _FpFormatI64\n");
    i64 = (((__int64)ieee->Result.Value.I64Value.W[1]) << 32) |
        (unsigned __int64)ieee->Result.Value.I64Value.W[0];
    printf ("USER HANDLER: result value = %I64x\n", i64);
  } else if (ieee->Result.Format == _FpFormatU32) {
    printf ("USER HANDLER: result has format _FpFormatU32\n");
    u32 = ieee->Result.Value.U32Value;
    printf ("USER HANDLER: result value = %8.8x\n", u32);
  } else if (ieee->Result.Format == _FpFormatU64) {
    printf ("USER HANDLER: result has format _FpFormatU64\n");
    u64 = (((unsigned __int64)ieee->Result.Value.U64Value.W[1]) << 32) |
        (unsigned __int64)ieee->Result.Value.U64Value.W[0];
    printf ("USER HANDLER: result value = %I64x\n", u64);
  } else if (ieee->Result.Format == _FpFormatCompare) {
    printf ("USER HANDLER: result has format _FpFormatCompare\n");
  } else {
    printf ("USER HANDLER: result has unknown format\n");
  }
} else {
  printf ("USER HANDLER: result is not valid\n");
}

// the result should be set according to the excepting operation, and the
// result format; in this example, it will be assumed that the result is a
// single precision floating-point number
if (ieee->Cause.InvalidOperation) {
  ieee->Result.Value.U32Value = 0x3f800001; /* 1.0 + 1 ulp */
  printf ("USER HANDLER: invalid exception\n");
} else if (ieee->Cause.ZeroDivide) {
  ieee->Result.Value.U32Value = 0x3f800002; /* 1.0 + 2 ulp */
  printf ("USER HANDLER: zero divide exception\n");
} else if (ieee->Cause.Overflow) {
  ieee->Result.Value.U32Value = 0x3f800003; /* 1.0 + 3 ulp */
  printf ("USER HANDLER: overflow exception\n");
} else if (ieee->Cause.Underflow) {
  ieee->Result.Value.U32Value = 0x3f800004; /* 1.0 + 4 ulp */
  printf ("USER HANDLER: underflow exception\n");
} else if (ieee->Cause.Inexact) {
```

```
    ieee->Result.Value.U32Value = 0x3f800005; /* 1.0 + 5 ulp */
    printf ("USER HANDLER: inexact exception\n");
  } else { /* CauseDenormal is assumed */
    ieee->Result.Value.U32Value = 0x3f800006; /* 1.0 + 6 ulp */
    printf ("USER HANDLER: denormal exception\n");
  }

  printf ("\n *** END USER HANDLER ***\n\n");

  return EXCEPTION_CONTINUE_EXECUTION;
}

// Example for raising and handling unmasked floating-point exceptions;
// an assembly coded function, run_fma (), executes fma.s.s0 f6 = f7, f8, f9,
// with pointers to the FPSR and to the single precision parameters passed by
// the caller (the single precision floating-point values are specified by
// their hexadecimal representation, as unsigned integers); passing the FPSR
// value is necessary in order to selectively unmask (enable) floating-point
// exceptions
main () {

  unsigned __int64 fpsr;
  unsigned int opd1, opd2, opd3, res;

  _try {

        // IMPORTANT NOTE: when setting the FPSR, always leave status field 1 (sf1)
        // at its default value (traps disabled, rounding-to-nearest, 64-bit
        // precision, widest range exponent set, flush-to-zero mode not set),
        // otherwise floating-point divide, square root, remainder, and libm
         // transcendental function calculations, as well as integer divide and
         // remainder operations might generate incorrect results
    /* invalid exception */
    printf ("\n****************** INVALID EXCEPTION ******************** \n");
    fpsr = 0x0009804c0270033e; /* default FPSR, with I exceptions unmasked */
    opd1 = 0x00000000; /* 0.0 */
    opd2 = 0x7f800000; /* +infinity */
    opd3 = 0x00000000; /* 0.0 */
    res = 0x00000000; /* 0.0 */
    run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
    printf ("after invalid exception res = %8.8x\n", res);

    /* denormal exception */
    printf ("\n****************** DENORMAL EXCEPTION ******************* \n");
    fpsr = 0x0009804c0270033d; /* default FPSR, with D exceptions unmasked */
    opd1 = 0x3f800000; /* 1.0 */
    opd2 = 0x00000001; /* smallest positive denormal */
    opd3 = 0x00000000; /* 0.0 */
    res = 0x00000000; /* 0.0 */
    run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
```

```
    printf ("after denormal exception res = %8.8x\n", res);

    /* Overflow exception */
    printf ("\n****************** OVERFLOW EXCEPTION ******************** \n");
    fpsr = 0x0009804c02700337; /* default FPSR, with O exceptions unmasked */
    opd1 = 0x7f7fffff; /* 1.1...1 * 2^127 */
    opd2 = 0x7f7fffff; /* 1.1...1 * 2^127 */
    opd3 = 0x00000000; /* 0.0 */
    res = 0x00000000; /* 0.0 */
    run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
    printf ("after Overflow exception res = %8.8x\n", res);

    /* Underflow exception */
    printf ("\n****************** UNDERFLOW EXCEPTION ******************** \n");
    fpsr = 0x0009804c0270032f; /* default FPSR, with U exceptions unmasked */
    opd1 = 0x00800000; /* smallest positive normal */
    opd2 = 0x00800000; /* smallest positive normal */
    opd3 = 0x00000000; /* 0.0 */
    res = 0x00000000; /* 0.0 */
    run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
    printf ("after Underflow exception res = %8.8x\n", res);

    /* inexact exception */
    printf ("\n****************** INEXACT EXCEPTION ******************** \n");
    fpsr = 0x0009804c0270031f; /* default FPSR, with P exceptions unmasked */
    opd1 = 0x3f800001; /* 1.0 + 1 ulp */
    opd2 = 0x3f800001; /* 1.0 + 1 ulp */
    opd3 = 0x00000000; /* 0.0 */
    res = 0x00000000; /* 0.0 */
    run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
    printf ("after inexact exception res = %8.8x\n", res);

} _except (_fpieee_flt (GetExceptionCode(), GetExceptionInformation(),
fpieee_handler)) {

    printf ("ERROR: handler returned EXCEPTION_EXECUTE_HANDLER");

                 }

}


run_fma:
  alloc r31 = ar.pfs,5,2,0,0  // r32, r33, r34, r35, r36, r37, r38

  // &fpsr is in r32
  // &res (output) is in r33
  // &opd1 (input) is in r34
  // &opd2 (input) is in r35
  // &opd3 (input) is in r36
```

```
mov r38 = ar40;; // save old FPSR in r38
ld8 r37 = [r32];; // load new FPSR in r37
mov ar40 = r37;; // set new value of FPSR
ldfs f7 = [r34] // load first input argument into f7
ldfs f8 = [r35] // load second input argument into f8
ldfs f9 = [r36];; // load third input argument into f9
fma.s.s0 f6 = f7, f8, f9;; // f6 = f7 * f8 + f9
mov r37 = ar40;; // store new FPSR
st8 [r32] = r37;;
stfs [r33] = f6 // store result
mov ar40 = r38;; // restore original FPSR
br.ret.sptk b0 // return
.endp run_fma
```

The output for this first example is:

```
****************** INVALID EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 1
USER HANDLER: enable bits PUOZI = 0 0 0 0 1
USER HANDLER: status bits PUOZI = 0 0 0 0 1
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 00000000000000000000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000001ffff8000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: invalid exception

 *** END USER HANDLER ***

after invalid exception res = 3f800001

****************** DENORMAL EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
```

```
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 0
USER HANDLER: enable bits PUOZI = 0 0 0 0 0
USER HANDLER: status bits PUOZI = 0 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000000ff810000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: denormal exception

 *** END USER HANDLER ***

after denormal exception res = 3f800006

****************** OVERFLOW EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 1 0 0
USER HANDLER: enable bits PUOZI = 0 0 1 0 0
USER HANDLER: status bits PUOZI = 1 0 1 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000001007efffff0000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000001007efffff0000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 5f7ffffe
USER HANDLER: overflow exception

 *** END USER HANDLER ***

after Overflow exception res = 3f800003

****************** UNDERFLOW EXCEPTION ********************
```

```
 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 1 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 1 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ff818000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000000ff818000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 21800000
USER HANDLER: underflow exception

 *** END USER HANDLER ***

after Underflow exception res = 3f800004

****************** INEXACT EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 1 0 0 0 0
USER HANDLER: enable bits PUOZI = 1 0 0 0 0
USER HANDLER: status bits PUOZI = 1 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ffff8000010000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000000ffff8000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 3f800002
USER HANDLER: inexact exception
```

```
 *** END USER HANDLER ***

after inexact exception res = 3f800005
```

3.2. Example 2. Floating-Point Exceptions Raised by Scalar Instructions from C Language Code

The second example illustrates usage of the IEEE filter for handling unmasked floating-point exceptions raised by scalar Itanium processor instructions, directly from C language code. The user exception handler is the same as in Example 1. The output is also the same. The contents of `main()` are different, and a new function, `write_fpsr()` is used to unmask floating-point exceptions:

```
main ()  {
  unsigned __int64 FPSR;
  unsigned int opd1, opd2, res;
  float fopd1, fopd2, fres;

  // Example for raising unmasked floating-point exceptions from a C program;
  // the FPSR is written in order to selectively unmask (enable) floating-point
  // exceptions
  _try {

    // IMPORTANT NOTE: when setting the FPSR, always leave status field 1 (sf1)
    // at its default value (traps disabled, rounding-to-nearest, 64-bit
    // precision, widest range exponent set, flush-to-zero mode not set),
    // otherwise floating-point divide, square root, remainder, and libm
    // transcendental function calculations, as well as integer divide and
    // remainder operations might generate incorrect results

    /* invalid exception */
    printf ("\n****************** INVALID EXCEPTION ******************** \n");
    FPSR = 0x0009804c0270033e; /* default FPSR, with I exceptions unmasked */
    opd1 = 0x00000000;
    fopd1 = *(float *)&opd1; /* 0.0 */
    opd2 = 0x7f800000;
    fopd2= *(float *)&opd2; /* +infinity */
    res = 0x00000000;
    fres = *(float *)&res; /* 0.0 */
    write_fpsr (FPSR);
    fres = fopd1 * fopd2;
    res = *(unsigned int *)&fres;
    printf ("after invalid exception res = %8.8x\n", res);

    /* denormal exception */
    printf ("\n****************** DENORMAL EXCEPTION ******************** \n");
    FPSR = 0x0009804c0270033d; /* default FPSR, with D exceptions unmasked */
    opd1 = 0x3f800000;
```

```
   fopd1 = *(float *)&opd1; /* 1.0 */
   opd2 = 0x00000001;
   fopd2= *(float *)&opd2; /* smallest positive denormal */
   res = 0x00000000;
   fres = *(float *)&res; /* 0.0 */
   write_fpsr (FPSR);
   fres = fopd1 * fopd2;
   res = *(unsigned int *)&fres;
   printf ("after denormal exception res = %8.8x\n", res);

   /* Overflow exception */
   printf ("\n****************** OVERFLOW EXCEPTION ******************** \n");
   FPSR = 0x0009804c02700337; /* default FPSR, with O exceptions unmasked */
   opd1 = 0x7f7fffff;
   fopd1 = *(float *)&opd1; /* 1.1...1 * 2^127 */
   opd2 = 0x7f7fffff;
   fopd2 = *(float *)&opd2; /* 1.1...1 * 2^127 */
   res = 0x00000000;
   fres = *(float *)&res; /* 0.0 */
   write_fpsr (FPSR);
   fres = fopd1 * fopd2;
   res = *(unsigned int *)&fres;
   printf ("after Overflow exception res = %8.8x\n", res);

   /* Underflow exception */
   printf ("\n****************** UNDERFLOW EXCEPTION ******************** \n");
   FPSR = 0x0009804c0270032f; /* default FPSR, with U exceptions unmasked */
   opd1 = 0x00800000;
   fopd1 = *(float *)&opd1; /* smallest positive normal */
   opd2 = 0x00800000;
   fopd2 = *(float *)&opd2; /* smallest positive normal */
   res = 0x00000000;
   fres = *(float *)&res; /* 0.0 */
   write_fpsr (FPSR);
fres = fopd1 * fopd2;
res = *(unsigned int *)&fres;
printf ("after Underflow exception res = %8.8x\n", res);

/* inexact exception */
printf ("\n****************** INEXACT EXCEPTION ******************** \n");
   FPSR = 0x0009804c0270031f; /* default FPSR, with P exceptions unmasked */
   opd1 = 0x3f800001;
   fopd1 = *(float *)&opd1; /* 1.0 + 1 ulp */
   opd2 = 0x3f800001;
   fopd2 = *(float *)&opd2; /* 1.0 + 1 ulp */
   res = 0x00000000;
   fres = *(float *)&res; /* 0.0 */
   write_fpsr (FPSR);
   fres = fopd1 * fopd2;
   res = *(unsigned int *)&fres;
   printf ("after inexact exception res = %8.8x\n", res);
```

```
  } _except (_fpieee_flt (GetExceptionCode(), GetExceptionInformation(),
      fpieee_handler)) {
    printf ("ERROR: handler returned EXCEPTION_EXECUTE_HANDLER");
  }
}


write_fpsr:
  alloc r31 = ar.pfs,1,0,0,0  // r32

  // fpsr is in r32
  mov ar40 = r32;;
  br.ret.sptk b0;; // return
  .endp write_fpsr
```

3.3. Example 3. Floating-Point Exceptions Raised by Parallel Instructions from Assembly Language Code

The third example illustrates the usage of the IEEE filter for handling unmasked floating-point exceptions raised by parallel (SIMD) Itanium processor instructions. The user exception handler is the same as in Example 1, since the IEEE filter presents the exceptions individually to the user handler, one at a time, even if more than one exception per instruction occurs (so the user does not need to provide special code for handling exceptions raised by parallel instructions). Function `main()` calls this time `run_fpma()`, listed next, which is used to trigger a few of the possible exception combinations from the `fpma` instruction: invalid operation in the low half, overflow in the high half, underflow in the high half combined with denormal operand in the low half, and finally denormal operand in the high half combined with underflow in the low half. The last two cases are symmetric, to illustrate that processing of simultaneous floating-point exceptions is performed by the IEEE filter in the order low first, and then high.

```
main () {

unsigned __int64 FPSR;
_FP128 opd1, opd2, opd3, res;

_try {

  /* (no exception, invalid exception) */
  printf ("\n************** LOW HALF INVALID EXCEPTION *************** \n");
  FPSR = 0x0009804c0270033e; /* default FPSR, with I exceptions unmasked */
  opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
  opd1.W[1] = 0x3f800000; opd1.W[0] = 0x3f800000; /* (1.0, 1.0) */
  opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
  opd2.W[1] = 0x3f800000; opd2.W[0] = 0x7f800000; /* (1.0, +infinity) */
  opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
  opd3.W[1] = 0x3f800000; opd3.W[0] = 0xff800000; /* (1.0, -infinity) */
  res.W[3] = 0x0;  res.W[2] = 0x01003e;
  res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
  run_fpma (&FPSR, &res, &opd1, &opd2, &opd3);
```

```
printf ("after low invalid exception res = %8.8x %8.8x %8.8x %8.8x\n",
    res.W[3], res.W[2], res.W[1], res.W[0]);

/* (overflow exception, no exception) */
printf ("\n************** HIGH HALF OVERFLOW EXCEPTION ************** \n");
FPSR = 0x0009804c02700337; /* default FPSR, with O exceptions unmasked */
opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
opd1.W[1] = 0x7f7fffff; opd1.W[0] = 0x3f800000; /* (1.1...1 * 2^127, 1.0) */
opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
opd2.W[1] = 0x7f7fffff; opd2.W[0] = 0x3f800000; /* (1.1...1 * 2^127, 1.0) */
opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
opd3.W[1] = 0x3f800000; opd3.W[0] = 0x00000000; /* (1.0, 0.0) */
res.W[3] = 0x0;  res.W[2] = 0x01003e;
res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
run_fpma (&FPSR, &res, &opd1, &opd2, &opd3);
printf ("after high overflow exception res = %8.8x %8.8x %8.8x %8.8x\n",
    res.W[3], res.W[2], res.W[1], res.W[0]);

/* (underflow exception, denormal exception) */
printf ("\n**** HIGH HALF UNDERFLOW, LOW HALF DENORMAL EXCEPTION **** \n");
FPSR = 0x0009804c0270032d; /* default FPSR, with U, D exceptions unmasked */
opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
opd1.W[1] = 0x00800000; opd1.W[0] = 0x3f800000; /* (smallest normal, 1.0) */
opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
opd2.W[1] = 0x00800000; opd2.W[0] = 0x00000001;
    /* (smallest normal, smallest denormal) */
opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
opd3.W[1] = 0x00000000; opd3.W[0] = 0x00000000; /* (0.0, 0.0) */
res.W[3] = 0x0;  res.W[2] = 0x01003e;
res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
run_fpma (&FPSR, &res, &opd1, &opd2, &opd3);
printf ("after high underflow & low denormal exception res = “
  “%8.8x %8.8x %8.8x %8.8x\n", res.W[3], res.W[2], res.W[1], res.W[0]);

/* (denormal exception, underflow exception) */
printf ("\n**** HIGH HALF DENORMAL, LOW HALF UNDERFLOW EXCEPTION **** \n");
FPSR = 0x0009804c0270032d; /* default FPSR, with U, D exceptions unmasked */
opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
opd1.W[1] = 0x3f800000; opd1.W[0] = 0x00800000; /* (1.0, smallest normal) */
opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
opd2.W[1] = 0x00000001; opd2.W[0] = 0x00800000;
    /* (smallest denormal, smallest normal) */
opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
opd3.W[1] = 0x00000000; opd3.W[0] = 0x00000000; /* (0.0, 0.0) */
res.W[3] = 0x0;  res.W[2] = 0x01003e;
res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
run_fpma (&FPSR, &res, &opd1, &opd2, &opd3);
printf ("after high denormal & low underflow exception res = “
    “%8.8x %8.8x %8.8x %8.8x\n", res.W[3], res.W[2], res.W[1], res.W[0]);
```

```
  } _except (_fpieee_flt (GetExceptionCode(), GetExceptionInformation(),
      fpieee_handler)) {

    printf ("ERROR: handler returned EXCEPTION_EXECUTE_HANDLER");

  }

  }


run_fpma:
  alloc r31 = ar.pfs,5,2,0,0  // r32, r33, r34, r35, r36, r37, r38
  // &fpsr is in r32
  // &res (output) is in r33
  // &opd1 (input) is in r34
  // &opd2 (input) is in r35
  // &opd3 (input) is in r36

  mov r38 = ar40;; // save old FPSR in r38
  ld8 r37 = [r32];; // load new FPSR in r37
  mov ar40 = r37;; // set new value of FPSR
  ldf.fill f7 = [r34] // load first input argument into f7
  ldf.fill f8 = [r35] // load second input argument into f8
  ldf.fill f9 = [r36];; // load third input argument into f9
  fpma.s0 f6 = f7, f8, f9;; // f6 = f7 * f8 + f9
  mov r37 = ar40;; // store new FPSR
  st8 [r32] = r37;;
  stf.spill [r33] = f6 // store result
  mov ar40 = r38;; // restore original FPSR
  br.ret.sptk b0 // return
  .endp run_fpma
```

The output for this third example is:

```
************** LOW HALF INVALID EXCEPTION ****************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 1
USER HANDLER: enable bits PUOZI = 0 0 0 0 1
USER HANDLER: status bits PUOZI = 0 0 0 0 1
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ffff8000000000000000
```

```
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000001ffff8000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 000000000003ffff8000000000000000
USER HANDLER: result is not valid
USER HANDLER: invalid exception

 *** END USER HANDLER ***

after low invalid exception res = 00000000 0001003e 40000000 3f800001

************** HIGH HALF OVERFLOW EXCEPTION **************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 1 0 0
USER HANDLER: enable bits PUOZI = 0 0 1 0 0
USER HANDLER: status bits PUOZI = 1 0 1 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000001007effffff0000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000001007effffff0000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 000000000000ffff8000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 1f7ffffe
USER HANDLER: overflow exception

 *** END USER HANDLER ***

after high overflow exception res = 00000000 0001003e 3f800003 3f800000

**** HIGH HALF UNDERFLOW, LOW HALF DENORMAL EXCEPTION ****

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
```

```
USER HANDLER: status bits PUOZI = 0 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 00000000000000000000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: denormal exception

 *** END USER HANDLER ***

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 1 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 1 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ff818000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000000ff818000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 61800000
USER HANDLER: underflow exception

 *** END USER HANDLER ***

after high underflow & low denormal exception res = 00000000 0001003e 3f800004 3f800006

**** HIGH HALF DENORMAL, LOW HALF UNDERFLOW EXCEPTION ****

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 1 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
```

```
USER HANDLER: status bits PUOZI = 0 1 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ff818000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 000000000000ff818000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _FpFormatFp32
USER HANDLER: result value = 61800000
USER HANDLER: underflow exception

 *** END USER HANDLER ***


 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double-extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _FpFormatFp82
USER HANDLER: operand1 value = 000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _FpFormatFp82
USER HANDLER: operand2 value = 00000000000000000000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _FpFormatFp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: denormal exception

 *** END USER HANDLER ***

after high denormal & low underflow exception res = 00000000 0001003e 3f800006 3f800004
```

4. Authors

Marius Cornea (marius.cornea@intel.com)

5. Acknowledgments

Bob Norin and Asit Mallick had a significant contribution in creating this document.

6. References

[1] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985.

[2] Intel Corporation, *The Itanium™ Architecture Software Developer's Manual*, at http://developer.intel.com/design/ia-64/manuals

[3] Cornea-Hasegan, M. and Norin, B., *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic*, Intel Technology Journal, Q4, 1999, at http://developer.intel.com/technology/itj/q41999.htm

[4] Intel Corporation, *Itanium™ Processor Floating-point Software Assistance and Floating-point Exception Handling*, at http://developer.intel.com/software/products/opensource/libraries/num.htm